19 | 土地不能浪费：如何管理内存对象？

2021-06-21 LMOS

《操作系统实战45讲》

课程介绍 >

讲述：陈晨

时长 15:07　大小 13.86M

你好，我是 LMOS。

在前面的课程中，我们建立了物理内存页面管理器，它既可以分配单个页面，也可以分配多个连续的页面，还能指定在特殊内存地址区域中分配页面。

但你发现没有，物理内存页面管理器一次分配至少是一个页面，而我们对内存分页是一个页面 4KB，即 4096 字节。对于小于一个页面的内存分配请求，它无能为力。如果要实现小于一个页面的内存分配请求，又该怎么做呢？

这节课我们就一起来解决这个问题。课程配套代码，你可以从这里获得。

malloc 给我们的启发

首先，我想和你说说，为什么小于一个页面的内存我们也要格外珍惜？

如果你在大学学过 C 程序设计语言的话，相信你对 C 库中的 malloc 函数也不会陌生，它负责完成分配一块内存空间的功能。

下面的代码。我相信你也写过，或者写过类似的，不用多介绍你也可以明白。

复制代码

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main() {
    char *str;
    //内存分配 存放15个char字符类型
    str = (char *) malloc(15);
    if (str == NULL) {
        printf("mem alloc err\n");
        return -1;
    }
    //把hello world字符串复制到str开始的内存地址空间中
    strcpy(str, "hello world");
    //打印hello world字符串和它的地址
    printf("String = %s,  Address = %u\n", str, str);
    //释放分配的内存
    free(str);
    return(0);
}
```

这个代码流程很简单，就是分配一块 15 字节大小的内存空间，然后把字符串复制到分配的内存空间中，最后用字符串的形式打印了那个块内存，最后释放该内存空间。

但我们并不是要了解 malloc、free 函数的工作原理，而是要清楚，像这样分配几个字节内存空间的操作，这在内核中比比皆是。

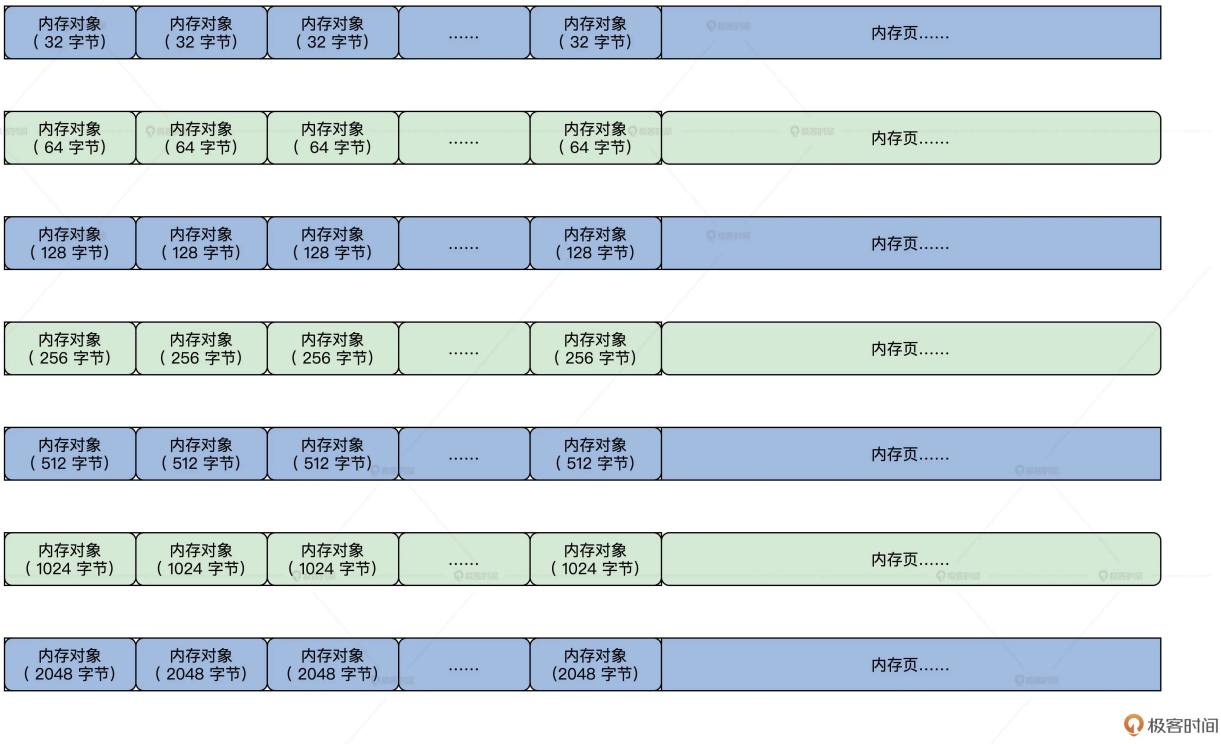
页还能细分吗

是的，单从内存角度来看，页最小是以字节为单位的。但是从 MMU 角度看，内存是以页为单位的，所以我们的 Cosmos 的物理内存分配器也以页为单位。现在的问题是，内核中有大量远小于一个页面的内存分配请求，如果对此还是分配一个页面，就会浪费内存。

要想解决这个问题，**就要细分“页”这个单位。**虽然从 MMU 角度来看，页不能细分，但是从软件逻辑层面页可以细分，但是如何分，则十分讲究。

结合历史经验和硬件特性（Cache 行大小）来看，我们可以把一个页面或者连续的多个页面，分成 32 字节、64 字节、128 字节、256 字节、512 字节、1024 字节、2048 字节、4096 字节（一个页）。这些都是 Cache 行大小的倍数。我们给这些小块内存取个名字，叫**内存对象。**

我们可以这样设计：**把一个或者多个内存页面分配出来，作为一个内存对象的容器，在这个容器中容纳相同的内存对象，即同等大小的内存块。**你可以把这个容器，想像成一个内存对象数组。为了让你更好理解，我还给你画了张图解释。



内存对象视图

如何表示一个内存对象

前面只是进行了理论上的设计和构想，下面我们就通过代码来实现这些构想，真正把想法变成现实。

我们从内存对象开始入手。如何表示一个内存对象呢？当然是要设计一个表示内存对象的数据结构，代码如下所示：

复制代码

```
typedef struct s_FREOBJH
{
    list_h_t oh_list;     //链表
    uint_t oh_stus;       //对象状态
    void* oh_stat;        //对象的开始地址
}freobjh_t;
```

我们在后面的代码中就用 freobjh_t 结构表示一个对象，其中的链表是为了找到这个对象。是不是很简单？没错，表示一个内存对象就是如此简单。

内存对象容器

光有内存对象还不够，如何放置内存对象是很重要的。根据前面的构想，为了把多个同等大小的内存对象放在一个内存对象容器中，我们需要设计出表示内存对象容器的数据结构。内存容器要占用内存页面，需要内存对象计数信息、内存对象大小信息，还要能扩展容量。

把上述功能综合起来，代码如下所示。

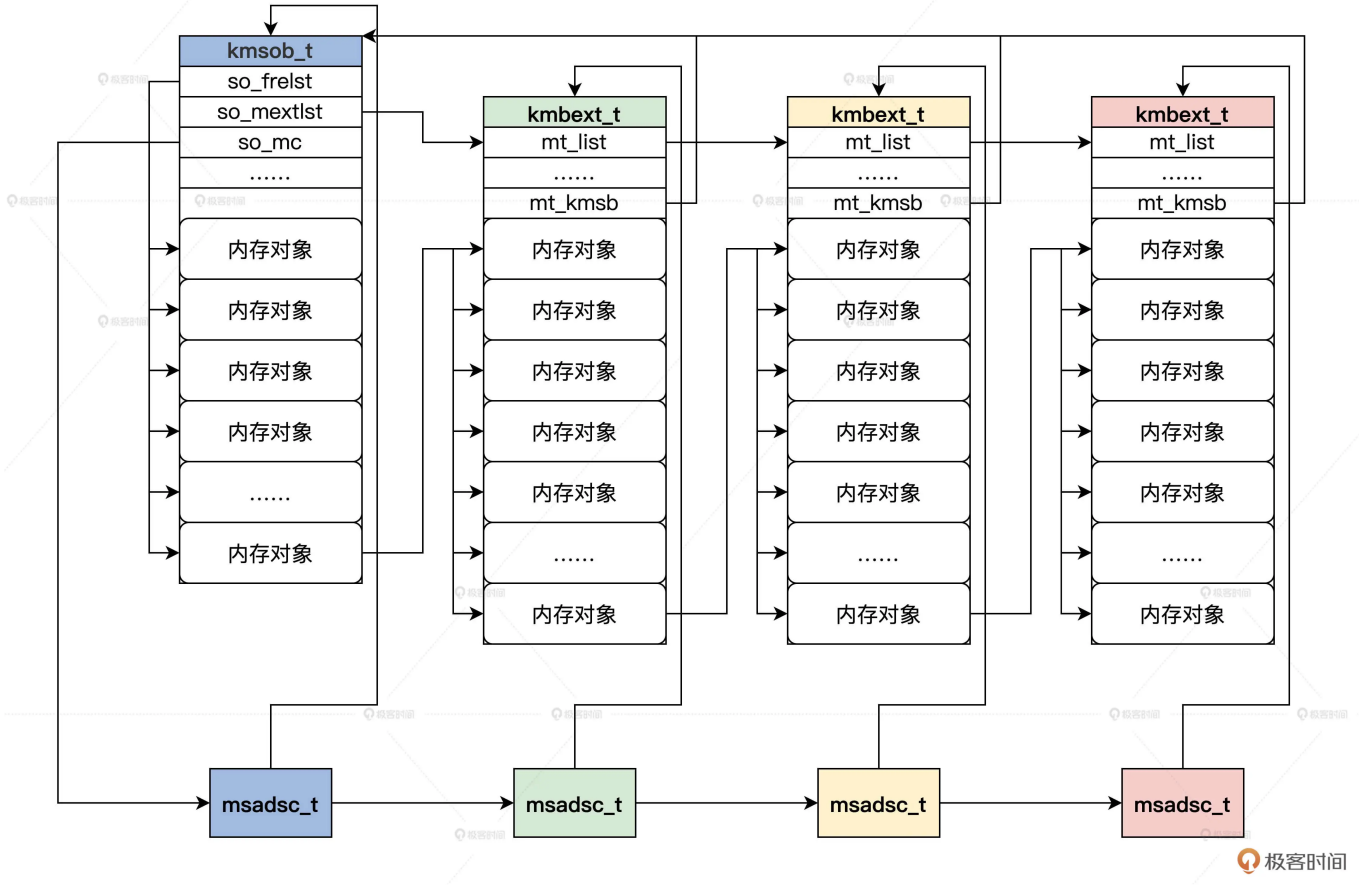
复制代码

```
//管理内存对象容器占用的内存页面所对应的msadsc_t结构
typedef struct s_MSCLST
{
    uint_t ml_msanr;  //多少个msadsc_t
    uint_t ml_ompnr;  //一个msadsc_t对应的连续的物理内存页面数
    list_h_t ml_list; //挂载msadsc_t的链表
}msclst_t;
//管理内存对象容器占用的内存
typedef struct s_MSOMDC
{
    //msclst_t结构数组mc_lst[0]=1个连续页面的msadsc_t
    //               mc_lst[1]=2个连续页面的msadsc_t
    //               mc_lst[2]=4个连续页面的msadsc_t
    //               mc_lst[3]=8个连续页面的msadsc_t
    //               mc_lst[4]=16个连续页面的msadsc_t
    msclst_t mc_lst[MSCLST_MAX];
    uint_t mc_msanr;   //总共多个msadsc_t结构
    list_h_t mc_list;
    //内存对象容器第一个占用msadsc_t
    list_h_t mc_kmobinlst;
    //内存对象容器第一个占用msadsc_t对应的连续的物理内存页面数
    uint_t mc_kmobinpnr;
}msomdc_t;
```

```
//管理内存对象容器扩展容量
typedef struct s_KMBEXT
{
    list_h_t mt_list;         //链表
    adr_t mt_vstat;           //内存对象容器扩展容量开始地址
    adr_t mt_vend;            //内存对象容器扩展容量结束地址
    kmsob_t* mt_kmsb;         //指向内存对象容器结构
    uint_t mt_mobjnr;         //内存对象容器扩展容量的内存中有多少对象
}kmbext_t;
//内存对象容器
typedef struct s_KMSOB
{
    list_h_t so_list;         //链表
    spinlock_t so_lock;       //保护结构自身的自旋锁
    uint_t so_stus;           //状态与标志
    uint_t so_flgs;
    adr_t so_vstat;           //内存对象容器的开始地址
    adr_t so_vend;            //内存对象容器的结束地址
    size_t so_objsz;          //内存对象大小
    size_t so_objrelsz;       //内存对象实际大小
    uint_t so_mobjnr;         //内存对象容器中总共的对象个数
    uint_t so_fobjnr;         //内存对象容器中空闲的对象个数
    list_h_t so_frelst;       //内存对象容器中空闲的对象链表头
    list_h_t so_alclst;       //内存对象容器中分配的对象链表头
    list_h_t so_mextlst;      //内存对象容器扩展kmbext_t结构链表头
    uint_t so_mextnr;         //内存对象容器扩展kmbext_t结构个数
    msomdc_t so_mc;           //内存对象容器占用内存页面管理结构
    void* so_privp;           //本结构私有数据指针
    void* so_extdp;           //本结构扩展数据指针
}kmsob_t;
```

这段代码中设计了四个数据结构：kmsob_t 用于表示内存对象容器，kmbext_t 用于表示内存对象容器的扩展内存，msomdc_t 和 msclst_t 用于管理内存对象容器占用的物理内存页面。

你可能很难理解它们之间的关系，所以我为你准备了一幅图，如下所示。



内存对象容器关系

结合图示我们可以发现，在一组连续物理内存页面（用来存放内存对象）的开始地址那里，就存放着我们 kmsob_t 和 kmbext_t 的实例变量，它们占用了几十字节的空间。

初始化

因为 kmsob_t、kmbext_t、freobjh_t 结构的实例变量，它们是建立内存对象容器时创建并初始化的，这个过程是伴随着分配内存对象而进行的，所以内存对象管理器的初始化很简单。

但是有一点还是要初始化的，那就是**管理 kmsob_t 结构的数据结构**，它用于挂载不同大小的内存容器。现在我们就在 cosmos/hal/x86/ 目录下建立一个 kmsob.c 文件，来实现这个数据结构并初始化，代码如下所示。

复制代码

```
#define KOBLST_MAX (64)
//挂载kmsob_t结构
typedef struct s_KOBLST
{
    list_h_t ol_emplst; //挂载kmsob_t结构的链表
```

```
    kmsob_t* ol_cahe;    //最近一次查找的kmsob_t结构
    uint_t ol_emnr;      //挂载kmsob_t结构的数量
    size_t ol_sz;        //kmsob_t结构中内存对象的大小
}koblst_t;
//管理kmsob_t结构的数据结构
typedef struct s_KMSOBMGRHED
{
    spinlock_t ks_lock;  //保护自身的自旋锁
    list_h_t ks_tclst;   //链表
    uint_t ks_tcnr;
    uint_t ks_msobnr;    //总共多少个kmsob_t结构
    kmsob_t* ks_msobche; //最近分配内存对象的kmsob_t结构
    koblst_t ks_msoblst[KOBLST_MAX]; //koblst_t结构数组
}kmsobmgrhed_t;
//初始化koblst_t结构体
void koblst_t_init(koblst_t *initp, size_t koblsz)
{
    list_init(&initp->ol_emplst);
    initp->ol_cahe = NULL;
    initp->ol_emnr = 0;
    initp->ol_sz = koblsz;
    return;
}
//初始化kmsobmgrhed_t结构体
void kmsobmgrhed_t_init(kmsobmgrhed_t *initp)
{
    size_t koblsz = 32;
    knl_spinlock_init(&initp->ks_lock);
    list_init(&initp->ks_tclst);
    initp->ks_tcnr = 0;
    initp->ks_msobnr = 0;
    initp->ks_msobche = NULL;
    for (uint_t i = 0; i < KOBLST_MAX; i++)
    {
        koblst_t_init(&initp->ks_msoblst[i], koblsz);
        koblsz += 32;//这里并不是按照开始的图形分类的而是每次增加32字节，所以是32，64,9
    }
    return;
}
//初始化kmsob
void init_kmsob()
{
    kmsobmgrhed_t_init(&memmgrob.mo_kmsobmgr);
    return;
}
```

上面的代码注释已经很清楚了，就是 init_kmsob 函数调用 kmsobmgrhed_t_init 函数，在其中循环初始化 koblst_t 结构体数组，不多做解释。

但是有一点我们要搞清楚：**kmsobmgrhed_t 结构的实例变量是放在哪里的，它其实放在我们之前的 memmgrob_t 结构中了**，代码如下所示。

复制代码

```
//cosmos/include/halinc/halglobal.c
HAL_DEFGLOB_VARIABLE(memmgrob_t,memmgrob);

typedef struct s_MEMMGROB
{
    list_h_t mo_list;
    spinlock_t mo_lock;
    uint_t mo_stus;
    uint_t mo_flgs;
    //略去很多字段
    //管理kmsob_t结构的数据结构
    kmsobmgrhed_t mo_kmsobmgr;
    void* mo_privp;
    void* mo_extp;
}memmgrob_t;
//cosmos/hal/x86/memmgrinit.c
void init_memmgr()
{
    //初始化内存页结构
    init_msadsc();
    //初始化内存区结构
    init_memarea();
    //处理内存占用
    init_search_krloccupymm(&kmachbsp);
    //合并内存页到内存区中
    init_memmgrob();
    //初始化kmsob
    init_kmsob();
    return;
}
```

这并没有那么难，是不是？到这里，我们在内存管理初始化 init_memmgr 函数中调用了 init_kmsob 函数，对管理内存对象容器的结构进行了初始化，这样后面我们就能分配内存对象了。

分配内存对象

根据前面的初始化过程，我们只是初始化了 kmsobmgrhed_t 结构，却没初始化任何 kmsob_t 结构，而这个结构就是存放内存对象的容器，没有它是不能进行任何分配内存对象的操作的。

下面我们一起在分配内存对象的过程中探索，应该如何查找、建立 kmsob_t 结构，然后在 kmsob_t 结构中建立 freobjh_t 结构，最后在内存对象容器的容量不足时，一起来扩展容器的内存。

分配内存对象的接口

分配内存对象的流程，仍然要从分配接口开始。分配内存对象的接口很简单，只有一个内存对象大小的参数，然后返回内存对象的首地址。下面我们先在 kmsob.c 文件中写好这个函数，代码如下所示。

复制代码

```
//分配内存对象的核心函数
void *kmsob_new_core(size_t msz)
{
    //获取kmsobmgrhed_t结构的地址
    kmsobmgrhed_t *kmobmgrp = &memmgrob.mo_kmsobmgr;
    void *retptr = NULL;
    koblst_t *koblp = NULL;
    kmsob_t *kmsp = NULL;
    cpuflg_t cpuflg;
    //对kmsobmgrhed_t结构加锁
    knl_spinlock_cli(&kmobmgrp->ks_lock, &cpuflg);
    koblp = onmsz_retn_koblst(kmobmgrp, msz);
    if (NULL == koblp)
    {
        retptr = NULL;
        goto ret_step;
    }
    kmsp = onkoblst_retn_newkmsob(koblp, msz);
    if (NULL == kmsp)
    {
        kmsp = _create_kmsob(kmobmgrp, koblp, koblp->ol_sz);
        if (NULL == kmsp)
        {
            retptr = NULL;
            goto ret_step;
        }
    }
    retptr = kmsob_new_onkmsob(kmsp, msz);
    if (NULL == retptr)
    {
        retptr = NULL;
        goto ret_step;
    }
    //更新kmsobmgrhed_t结构的信息
    kmsob_updata_cache(kmobmgrp, koblp, kmsp, KUC_NEWFLG);
ret_step:
```

```
    //解锁kmsobmgrhed_t结构
    knl_spinunlock_sti(&kmobmgrp->ks_lock, &cpuflg);
    return retptr;
}
//内存对象分配接口
void *kmsob_new(size_t msz)
{
    //对于小于1 或者 大于2048字节的大小不支持 直接返回NULL表示失败
    if (1 > msz || 2048 < msz)
    {
        return NULL;
    }
    //调用核心函数
    return kmsob_new_core(msz);
}
```

上面代码中，内存对象分配接口很简单，只是对分配内存对象的大小进行检查，然后调用分配内存对象的核心函数，在这个核心函数中，就是围绕我们之前定义的几个数据结构，去进行一系列操作了。

但是究竟做了哪些操作呢，别急，我们继续往下看。

查找内存对象容器

根据前面的设计，我们已经知道内存对象是放在内存对象容器中的，所以要分配内存对象，必须要先根据要分配的内存对象大小，找到内存对象容器。

同时，我们还知道，内存对象容器数据结构 kmsob_t 就挂载在 kmsobmgrhed_t 数据结构中的 ks_msoblst 数组中，所以我们要遍历 ks_msoblst 数组，我们来写一个 onmsz_retn_koblst 函数，它返回 ks_msoblst 数组元素的指针，表示先根据内存对象的大小找到挂载 kmsob_t 结构对应的 koblst_t 结构。

复制代码

```
//看看内存对象容器是不是合乎要求
kmsob_t *scan_newkmsob_isok(kmsob_t *kmsp, size_t msz)
{
    //只要内存对象大小小于等于内存对象容器的对象大小就行
    if (msz <= kmsp->so_objsz)
    {
        return kmsp;
    }
    return NULL;
}
```

```
11
12 koblst_t *onmsz_retn_koblst(kmsobmgrhed_t *kmmgrhlokp, size_t msz)
13 {
14     //遍历ks_msoblst数组
15     for (uint_t kli = 0; kli < KOBLST_MAX; kli++)
16     {
17         //只要大小合适就返回
18         if (kmmgrhlokp->ks_msoblst[kli].ol_sz >= msz)
19         {
20             return &kmmgrhlokp->ks_msoblst[kli];
21         }
22     }
23     return NULL;
24 }
25
26 kmsob_t *onkoblst_retn_newkmsob(koblst_t *koblp, size_t msz)
27 {
28     kmsob_t *kmsp = NULL, *tkmsp = NULL;
29     list_h_t *tmplst = NULL;
30     //先看看上次分配所用到的koblst_t是不是正好是这次需要的
31     kmsp = scan_newkmsob_isok(koblp->ol_cahe, msz);
32     if (NULL != kmsp)
33     {
34         return kmsp;
35     }
36     //如果koblst_t中挂载的kmsob_t大于0
37     if (0 < koblp->ol_emnr)
38     {
39         //开始遍历koblst_t中挂载的kmsob_t
40         list_for_each(tmplst, &koblp->ol_emplst)
41         {
42             tkmsp = list_entry(tmplst, kmsob_t, so_list);
43             //检查当前kmsob_t是否合乎要求
44             kmsp = scan_newkmsob_isok(tkmsp, msz);
45             if (NULL != kmsp)
46             {
47                 return kmsp;
48             }
49         }
50     }
51     return NULL;
52 }
```

上述代码非常好理解，就是通过 onmsz_retn_koblst 函数，它根据内存对象大小查找并返回 ks_msoblst 数组元素的指针，这个数组元素中就挂载着相应的内存对象容器，然后由 onkoblst_retn_newkmsob 函数查询其中的内存对象容器并返回。

建立内存对象容器

不知道你发现没有，有一种情况必然会发生，那就是第一次分配内存对象时调用 onkoblst_retn_newkmsob 函数，它肯定会返回一个 NULL。因为第一次分配时肯定没有 kmsob_t 结构，所以我们在这个时候建立一个 kmsob_t 结构，即**建立内存对象容器。**

下面我们写一个 _create_kmsob 函数来创建 kmsob_t 结构，并执行一些初始化工作，代码如下所示。

复制代码

```
//初始化内存对象数据结构
void freobjh_t_init(freobjh_t *initp, uint_t stus, void *stat)
{
    list_init(&initp->oh_list);
    initp->oh_stus = stus;
    initp->oh_stat = stat;
    return;
}
//初始化内存对象容器数据结构
void kmsob_t_init(kmsob_t *initp)
{
    list_init(&initp->so_list);
    knl_spinlock_init(&initp->so_lock);
    initp->so_stus = 0;
    initp->so_flgs = 0;
    initp->so_vstat = NULL;
    initp->so_vend = NULL;
    initp->so_objsz = 0;
    initp->so_objrelsz = 0;
    initp->so_mobjnr = 0;
    initp->so_fobjnr = 0;
    list_init(&initp->so_frelst);
    list_init(&initp->so_alclst);
    list_init(&initp->so_mextlst);
    initp->so_mextnr = 0;
    msomdc_t_init(&initp->so_mc);
    initp->so_privp = NULL;
    initp->so_extdp = NULL;
    return;
}
//把内存对象容器数据结构，挂载到对应的koblst_t结构中去
bool_t kmsob_add_koblst(koblst_t *koblp, kmsob_t *kmsp)
{
    list_add(&kmsp->so_list, &koblp->ol_emplst);
    koblp->ol_emnr++;
    return TRUE;
}
//初始化内存对象容器
kmsob_t *_create_init_kmsob(kmsob_t *kmsp, size_t objsz, adr_t cvadrs, adr_t c
{
```

```
    //初始化kmsob结构体
    kmsob_t_init(kmsp);
    //设置内存对象容器的开始、结束地址，内存对象大小
    kmsp->so_vstat = cvadrs;
    kmsp->so_vend = cvadre;
    kmsp->so_objsz = objsz;
    //把物理内存页面对应的msadsc_t结构加入到kmsob_t中的so_mc.mc_kmobinlst链表上
    list_add(&msa->md_list, &kmsp->so_mc.mc_kmobinlst);
    kmsp->so_mc.mc_kmobinpnr = (uint_t)relpnr;
    //设置内存对象的开始地址为kmsob_t结构之后，结束地址为内存对象容器的结束地址
    freobjh_t *fohstat = (freobjh_t *)(kmsp + 1), *fohend = (freobjh_t *)cvadr

    uint_t ap = (uint_t)((uint_t)fohstat);
    freobjh_t *tmpfoh = (freobjh_t *)((uint_t)ap);
    for (; tmpfoh < fohend;)
    {//相当在kmsob_t结构体之后建立一个freobjh_t结构体数组
        if ((ap + (uint_t)kmsp->so_objsz) <= (uint_t)cvadre)
        {//初始化每个freobjh_t结构体
            freobjh_t_init(tmpfoh, 0, (void *)tmpfoh);
            //把每个freobjh_t结构体加入到kmsob_t结构体中的so_frelst中
           list_add(&tmpfoh->oh_list, &kmsp->so_frelst);
            kmsp->so_mobjnr++;
            kmsp->so_fobjnr++;
        }
        ap += (uint_t)kmsp->so_objsz;
        tmpfoh = (freobjh_t *)((uint_t)ap);
    }
    return kmsp;
}

//建立一个内存对象容器
kmsob_t *_create_kmsob(kmsobmgrhed_t *kmmgrlokp, koblst_t *koblp, size_t objsz
{
    kmsob_t *kmsp = NULL;
    msadsc_t *msa = NULL;
    uint_t relpnr = 0;
    uint_t pages = 1;
    if (128 < objsz)
    {
        pages = 2;
    }
    if (512 < objsz)
    {
        pages = 4;
    }
    //为内存对象容器分配物理内存空间，这是我们之前实现的物理内存页面管理器
    msa = mm_division_pages(&memmgrob, pages, &relpnr, MA_TYPE_KRNL, DMF_RELDI
    if (NULL == msa)
    {
        return NULL;
    }
    u64_t phyadr = msa->md_phyadrs.paf_padrs << PSHRSIZE;
```

```
    u64_t phyade = phyadr + (relpnr << PSHRSIZE) - 1;
    //计算它们的虚拟地址
    adr_t vadrs = phyadr_to_viradr((adr_t)phyadr);
    adr_t vadre = phyadr_to_viradr((adr_t)phyade);
    //初始化kmsob_t并建立内存对象
    kmsp = _create_init_kmsob((kmsob_t *)vadrs, koblp->ol_sz, vadrs, vadre, ms
    //把kmsob_t结构，挂载到对应的koblst_t结构中去
    if (kmsob_add_koblst(koblp, kmsp) == FALSE)
    {
        system_error(" _create_kmsob kmsob_add_koblst FALSE\n");
    }
    //增加计数
    kmmgrlokp->ks_msobnr++;
    return kmsp;
```

_create_kmsob 函数就是根据分配内存对象大小，建立一个内存对象容器。

首先，这个函数会找物理内存页面管理器申请一块连续内存页面。然后，在其中的开始部分建立 kmsob_t 结构的实例变量，又在 kmsob_t 结构的后面建立 freobjh_t 结构数组，并把每个 freobjh_t 结构挂载到 kmsob_t 结构体中的 so_frelst 中。最后再把 kmsob_t 结构，挂载到 kmsobmgrhed_t 结构对应的 koblst_t 结构中去。

上面的注释已经很清楚了，我相信你看得懂。

扩容内存对象容器

如果我们不断重复分配同一大小的内存对象，那么那个内存对象容器中的内存对象，迟早要分配完的。一旦内存对象分配完，内存对象容器就没有空闲的内存空间产生内存对象了。这时，我们就要为内存对象容器扩展内存空间了。

下面我们来写代码实现，如下所示。

复制代码

```
//初始化kmbext_t结构
void kmbext_t_init(kmbext_t *initp, adr_t vstat, adr_t vend, kmsob_t *kmsp)
{
    list_init(&initp->mt_list);
    initp->mt_vstat = vstat;
    initp->mt_vend = vend;
    initp->mt_kmsb = kmsp;
    initp->mt_mobjnr = 0;
    return;
```

```
}
//扩展内存页面
bool_t kmsob_extn_pages(kmsob_t *kmsp)
{
    msadsc_t *msa = NULL;
    uint_t relpnr = 0;
    uint_t pages = 1;
    if (128 < kmsp->so_objsz)
    {
        pages = 2;
    }
    if (512 < kmsp->so_objsz)
    {
        pages = 4;
    }
    //找物理内存页面管理器分配2或者4个连续的页面
    msa = mm_division_pages(&memmgrob, pages, &relpnr, MA_TYPE_KRNL, DMF_RELDI
    if (NULL == msa)
    {
        return FALSE;
    }
    u64_t phyadr = msa->md_phyadrs.paf_padrs << PSHRSIZE;
    u64_t phyade = phyadr + (relpnr << PSHRSIZE) - 1;
    adr_t vadrs = phyadr_to_viradr((adr_t)phyadr);
    adr_t vadre = phyadr_to_viradr((adr_t)phyade);
    //求出物理内存页面数对应在kmsob_t的so_mc.mc_lst数组中下标
    sint_t mscidx = retn_mscidx(relpnr);
    //把物理内存页面对应的msadsc_t结构加入到kmsob_t的so_mc.mc_lst数组中
    list_add(&msa->md_list, &kmsp->so_mc.mc_lst[mscidx].ml_list);
    kmsp->so_mc.mc_lst[mscidx].ml_msanr++;

    kmbext_t *bextp = (kmbext_t *)vadrs;
    //初始化kmbext_t数据结构
    kmbext_t_init(bextp, vadrs, vadre, kmsp);
//设置内存对象的开始地址为kmbext_t结构之后，结束地址为扩展内存页面的结束地址
    freobjh_t *fohstat = (freobjh_t *)(bextp + 1), *fohend = (freobjh_t *)vadr

    uint_t ap = (uint_t)((uint_t)fohstat);
    freobjh_t *tmpfoh = (freobjh_t *)((uint_t)ap);
    for (; tmpfoh < fohend;)
    {
        if ((ap + (uint_t)kmsp->so_objsz) <= (uint_t)vadre)
        {//在扩展的内存空间中建立内存对象
            freobjh_t_init(tmpfoh, 0, (void *)tmpfoh);
            list_add(&tmpfoh->oh_list, &kmsp->so_frelst);
            kmsp->so_mobjnr++;
            kmsp->so_fobjnr++;
            bextp->mt_mobjnr++;
        }
        ap += (uint_t)kmsp->so_objsz;
        tmpfoh = (freobjh_t *)((uint_t)ap);
    }
```

```
62      list_add(&bextp->mt_list, &kmsp->so_mextlst);
63      kmsp->so_mextnr++;
64      return TRUE;
65  `
```

有了前面建立内存对象容器的经验，加上这里的注释，我们理解上述代码并不难：不过是分配了另一块连续的内存空间，作为空闲的内存对象，并且把这块内存空间加内存对象容器中统一管理。

分配内存对象

有了内存对象容器，就可以分配内存对象了。由于我们前面精心设计了内存对象容器、内存对象等数据结构，这使得我们的内存对象分配代码时极其简单，而且性能极高。

下面我们来实现它吧！代码如下所示。

复制代码

```
1  //判断内存对象容器中有没有内存对象
2  uint_t scan_kmob_objnr(kmsob_t *kmsp)
3  {
4      if (0 < kmsp->so_fobjnr)
5      {
6          return kmsp->so_fobjnr;
7      }
8      return 0;
9  }
10 //实际分配内存对象
11 void *kmsob_new_opkmsob(kmsob_t *kmsp, size_t msz)
12 {
13     //获取kmsob_t中的so_frelst链表头的第一个空闲内存对象
14     freobjh_t *fobh = list_entry(kmsp->so_frelst.next, freobjh_t, oh_list);
15     //从链表中脱链
16     list_del(&fobh->oh_list);
17     //kmsob_t中的空闲对象计数减一
18     kmsp->so_fobjnr--;
19     //返回内存对象首地址
20     return (void *)(fobh);
21 }
22
23 void *kmsob_new_onkmsob(kmsob_t *kmsp, size_t msz)
24 {
25     void *retptr = NULL;
26     cpuflg_t cpuflg;
27     knl_spinlock_cli(&kmsp->so_lock, &cpuflg);
28     //如果内存对象容器中没有空闲的内存对象了就需要扩展内存对象容器的内存了
29     if (scan_kmsob_objnr(kmsp) < 1)
```

```
    {//扩展内存对象容器的内存
        if (kmsob_extn_pages(kmsp) == FALSE)
        {
            retptr = NULL;
            goto ret_step;
        }
    }
    //实际分配内存对象
    retptr = kmsob_new_opkmsob(kmsp, msz);
ret_step:
    knl_spinunlock_sti(&kmsp->so_lock, &cpuflg);
    return retptr;
}
```

分配内存对象的核心操作就是，**kmsob_new_opkmsob 函数从空闲内存对象链表头中取出第一个内存对象，返回它的首地址**。这个算法非常高效，无论内存对象容器中的内存对象有多少，kmsob_new_opkmsob 函数的操作始终是固定的，而如此高效的算法得益于我们先进的数据结构设计。

好了，到这里内存对象的分配就已经完成了，下面我们去实现内存对象的释放。

释放内存对象

释放内存对象，就是要把内存对象还给它所归属的内存对象容器。其逻辑就是根据释放内存对象的地址和大小，找到对应的内存对象容器，然后把该内存对象加入到对应内存对象容器的空闲链表上，最后看一看要不要释放内存对象容器占用的物理内存页面。

释放内存对象的接口

这里我们依然要从释放内存对象的接口开始实现，下面我们在 kmsob.c 文中写下这个函数，代码如下所示。

复制代码

```
bool_t kmsob_delete_core(void *fadrs, size_t fsz)
{
    kmsobmgrhed_t *kmobmgrp = &memmgrob.mo_kmsobmgr;
    bool_t rets = FALSE;
    koblst_t *koblp = NULL;
    kmsob_t *kmsp = NULL;
    cpuflg_t cpuflg;
    knl_spinlock_cli(&kmobmgrp->ks_lock, &cpuflg);
    //根据释放内存对象的大小在kmsobmgrhed_t中查找并返回koblst_t，在其中挂载着对应的kmsol
```

```
    koblp = onmsz_retn_koblst(kmobmgrp, fsz);
    if (NULL == koblp)
    {
        rets = FALSE;
        goto ret_step;
    }
    kmsp = onkoblst_retn_delkmsob(koblp, fadrs, fsz);
    if (NULL == kmsp)
    {
        rets = FALSE;
        goto ret_step;
    }
    rets = kmsob_delete_onkmsob(kmsp, fadrs, fsz);
    if (FALSE == rets)
    {
        rets = FALSE;
        goto ret_step;
    }
    if (_destroy_kmsob(kmobmgrp, koblp, kmsp) == FALSE)
    {
        rets = FALSE;
        goto ret_step;
    }
    rets = TRUE;
ret_step:
    knl_spinunlock_sti(&kmobmgrp->ks_lock, &cpuflg);
    return rets;
}
//释放内存对象接口
bool_t kmsob_delete(void *fadrs, size_t fsz)
{
    //对参数进行检查，但是多了对内存对象地址的检查
    if (NULL == fadrs || 1 > fsz || 2048 < fsz)
    {
        return FALSE;
    }
    //调用释放内存对象的核心函数
    return kmsob_delete_core(fadrs, fsz);
}
```

上述代码中，等到 kmsob_delete 函数检查参数通过之后，就调用释放内存对象的核心函数 kmsob_delete_core，在这个函数中，一开始根据释放内存对象大小，找到挂载其 kmsob_t 结构的 koblst_t 结构，接着又做了一系列的操作，这些操作正是我们接下来要实现的。

查找内存对象容器

释放内存对象，首先要找到这个将要释放的内存对象所属的内存对象容器。释放时的查找和分配时的查找不一样，因为要检查**释放的内存对象是不是属于该内存对象容器。**

下面我们一起来实现这个函数，代码如下所示。

复制代码

```
//检查释放的内存对象是不是在kmsob_t结构中
kmsob_t *scan_delkmsob_isok(kmsob_t *kmsp, void *fadrs, size_t fsz)
{//检查释放内存对象的地址是否落在kmsob_t结构的地址区间
    if ((adr_t)fadrs >= (kmsp->so_vstat + sizeof(kmsob_t)) && ((adr_t)fadrs +
    {    //检查释放内存对象的大小是否小于等于kmsob_t内存对象容器的对象大小
        if (fsz <= kmsp->so_objsz)
        {
            return kmsp;
        }
    }
    if (1 > kmsp->so_mextnr)
    {//如果kmsob_t结构没有扩展空间，直接返回
        return NULL;
    }
    kmbext_t *bexp = NULL;
    list_h_t *tmplst = NULL;
    //遍历kmsob_t结构中的每个扩展空间
    list_for_each(tmplst, &kmsp->so_mextlst)
    {
        bexp = list_entry(tmplst, kmbext_t, mt_list);
        //检查释放内存对象的地址是否落在扩展空间的地址区间
        if ((adr_t)fadrs >= (bexp->mt_vstat + sizeof(kmbext_t)) && ((adr_t)fad
        {//同样的要检查大小
            if (fsz <= kmsp->so_objsz)
            {
                return kmsp;
            }
        }
    }
    return NULL;
}
//查找释放内存对象所属的kmsob_t结构
kmsob_t *onkoblst_retn_delkmsob(koblst_t *koblp, void *fadrs, size_t fsz)
{
    v *kmsp = NULL, *tkmsp = NULL;
    list_h_t *tmplst = NULL;
    //看看上次刚刚操作的kmsob_t结构
    kmsp = scan_delkmsob_isok(koblp->ol_cahe, fadrs, fsz);
    if (NULL != kmsp)
    {
        return kmsp;
    }
    if (0 < koblp->ol_emnr)
```

```
    {    //遍历挂载koblp->ol_emplst链表上的每个kmsob_t结构
        list_for_each(tmplst, &koblp->ol_emplst)
        {
            tkmsp = list_entry(tmplst, kmsob_t, so_list);
            //检查释放的内存对象是不是属于这个kmsob_t结构
            kmsp = scan_delkmsob_isok(tkmsp, fadrs, fsz);
            if (NULL != kmsp)
            {
                return kmsp;
            }
        }
    }
    return NULL;
}
```

上面的代码注释已经很明白了，搜索对应 koblst_t 结构中的每个 kmsob_t 结构体，随后进行检查，检查了 kmsob_t 结构的自身内存区域和扩展内存区域。即比较释放内存对象的地址是不是落在它们的内存区间中，其大小是否合乎要求。

释放内存对象

如果不出意外，会找到释放内存对象的 kmsob_t 结构，这样就可以释放内存对象了，就是把这块内存空间还给内存对象容器，这个过程的具体代码实现如下所示。

复制代码

```
bool_t kmsob_del_opkmsob(kmsob_t *kmsp, void *fadrs, size_t fsz)
{
    if ((kmsp->so_fobjnr + 1) > kmsp->so_mobjnr)
    {
        return FALSE;
    }
    //让freobjh_t结构重新指向要释放的内存空间
    freobjh_t *obhp = (freobjh_t *)fadrs;
    //重新初始化块内存空间
    freobjh_t_init(obhp, 0, obhp);
    //加入kmsob_t结构的空闲链表
    list_add(&obhp->oh_list, &kmsp->so_frelst);
    //kmsob_t结构的空闲对象计数加一
    kmsp->so_fobjnr++;
    return TRUE;
}
//释放内存对象
bool_t kmsob_delete_onkmsob(kmsob_t *kmsp, void *fadrs, size_t fsz)
{
    bool_t rets = FALSE;
    cpuflg_t cpuflg;
```

```
    //对kmsob_t结构加锁
    knl_spinlock_cli(&kmsp->so_lock, &cpuflg);
    //实际完成内存对象释放
    if (kmsob_del_opkmsob(kmsp, fadrs, fsz) == FALSE)
    {
        rets = FALSE;
        goto ret_step;
    }
    rets = TRUE;
ret_step:
    //对kmsob_t结构解锁
    knl_spinunlock_sti(&kmsp->so_lock, &cpuflg);
    return rets;
}
```

结合上述代码和注释，我们现在明白了 kmsob_delete_onkmsob 函数调用 kmsob_del_opkmsob 函数。其核心机制就是**把要释放内存对象的空间，重新初始化，变成一个 freobjh_t 结构的实例变量，最后把这个 freobjh_t 结构加入到 kmsob_t 结构中空闲链表中**，这就实现了内存对象的释放。

销毁内存对象容器

如果我们释放了所有的内存对象，就会出现空的内存对象容器。如果下一次请求同样大小的内存对象，那么这个空的内存对象容器还能继续复用，提高性能。

但是你有没有想到，频繁请求的是不同大小的内存对象，那么空的内存对象容器会越来越多，这会占用大量内存，所以我们必须要把空的内存对象容器销毁。

下面我们写代码实现销毁内存对象容器。

复制代码

```
uint_t scan_freekmsob_isok(kmsob_t *kmsp)
{
    //当内存对象容器的总对象个数等于空闲对象个数时，说明这内存对象容器空闲
    if (kmsp->so_mobjnr == kmsp->so_fobjnr)
    {
        return 2;
    }
    return 1;
}

bool_t _destroy_kmsob_core(kmsobmgrhed_t *kmobmgrp, koblst_t *koblp, kmsob_t *
{
```

```
    list_h_t *tmplst = NULL;
    msadsc_t *msa = NULL;
    msclst_t *mscp = kmsp->so_mc.mc_lst;
    list_del(&kmsp->so_list);
    koblp->ol_emnr--;
    kmobmgrp->ks_msobnr--;
    //释放内存对象容器扩展空间的物理内存页面
    //遍历kmsob_t结构中的so_mc.mc_lst数组
    for (uint_t j = 0; j < MSCLST_MAX; j++)
    {
        if (0 < mscp[j].ml_msanr)
        {//遍历每个so_mc.mc_lst数组中的msadsc_t结构
            list_for_each_head_dell(tmplst, &mscp[j].ml_list)
            {
                msa = list_entry(tmplst, msadsc_t, md_list);
                list_del(&msa->md_list);
                //msadsc_t脱链
                //释放msadsc_t对应的物理内存页面
                if (mm_merge_pages(&memmgrob, msa, (uint_t)mscp[j].ml_ompnr) =
                {
                    system_error("_destroy_kmsob_core mm_merge_pages FALSE2\n"
                }
            }
        }
    }
    //释放内存对象容器本身占用的物理内存页面
    //遍历每个so_mc.mc_kmobinlst中的msadsc_t结构。它只会遍历一次
    list_for_each_head_dell(tmplst, &kmsp->so_mc.mc_kmobinlst)
    {
        msa = list_entry(tmplst, msadsc_t, md_list);
        list_del(&msa->md_list);
        //msadsc_t脱链
        //释放msadsc_t对应的物理内存页面
        if (mm_merge_pages(&memmgrob, msa, (uint_t)kmsp->so_mc.mc_kmobinpnr) =
        {
            system_error("_destroy_kmsob_core mm_merge_pages FALSE2\n");
        }
    }
    return TRUE;
}
//
```销毁内存对象容器
bool_t _destroy_kmsob(kmsobmgrhed_t *kmobmgrp, koblst_t *koblp, kmsob_t *kmsp)
{
    //看看能不能销毁
    uint_t screts = scan_freekmsob_isok(kmsp);
    if (2 == screts)
    {//调用销毁内存对象容器的核心函数
        return _destroy_kmsob_core(kmobmgrp, koblp, kmsp);
    }
    return FALSE;
}
```
```

上述代码中，首先会检查一下内存对象容器是不是空闲的，如果空闲，就调用**销毁内存对象容器的核心函数 _destroy_kmsob_core**。在 _destroy_kmsob_core 函数中，首先要释放内存对象容器的扩展空间所占用的物理内存页面，最后才可以释放内存对象容器自身占用物理内存页面。

请注意。**这个顺序不能前后颠倒**，这是因为扩展空间的物理内存页面对应的 msadsc_t 结构，它就挂载在 kmsob_t 结构的 so_mc.mc_lst 数组中。

好了，到这里我们内存对象释放的流程就完成了，这意味着我们整个内存对象管理也告一段落了。

重点回顾

今天我们从 malloc 函数入手，思考内核要怎样分配大量小块内存。我们把物理内存页面进一步细分成内存对象，为了表示和管理内存对象，又设计了内存对象、内存对象容器等一系列数据结构，随后写代码把它们初始化，最后我们依赖这些数据结构实现了内存对象管理算法。

下面我们来回顾一下这节课的重点。

1. 我们发现，在应用程序中可以使用 malloc 函数动态分配一些小块内存，其实这样的场景在内核中也是比比皆是。比如，内核经常要动态创建数据结构的实例变量，就需要分配小块的内存空间。

2. 为了实现内存对象的表示、分配和释放功能，我们定义了内存对象和内存对象容器的数据结构 freobjh_t、kmsob_t，并为了管理 kmsob_t 结构又定义了 kmsobmgrhed_t 结构。

3. 我们写好了初始化 kmsobmgrhed_t 结构的函数，并在 init_kmsob 中调用了它，进而又被 init_memmgr 函数调用，由于 kmsobmgrhed_t 结构是为了管理 kmsob_t 结构的所以在一开始就要被初始化。

4. 我们基于这些数据结构实现了内存对象的分配和释放。

思考题

为什么我们在分配内存对象大小时要按照 Cache 行大小的倍数分配呢？

欢迎你在留言区分享你的思考或疑问。如果这节课对你有帮助，也欢迎你分享给自己的同事、朋友，跟他一起交流讨论。

好，我是 LMOS，我们下节课见！

分享给需要的人，Ta订阅后你可得 20 元现金奖励

赞 3 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 18 | 划分土地（下）：如何实现内存页的分配与释放？

下一篇 20 | 土地需求扩大与保障：如何表示虚拟内存？

精选留言 (8)

写留言

pedro 置顶
2021-06-21

怕大家误会我的意思，我再说明一下。我上面的评论其实是在呼吁大家不要拘泥于代码，也不要深陷细节，一叶而障目是学习路上最容易犯的错误。
我以自身为例来说，初学数据结构时，链表是最简单的一种数据结构，老师一讲就懂，书一看就会，可是自己去写，尤其是链表的各种操作都是细节满满，很考验个人代码功底和思维完善性，至今我都记得从我懂链表起，到真正写出一个能work的链表都花了一周时...
展开

作者回复: 666666

11

neohope 置顶
2021-06-21

一、整理一下内存结构
1、memmgrob_t中有kmsobmgrhed_t
2、kmsobmgrhed_t中有一个koblst_t数组【KOBLST_MAX个】，序号为n的koblst_t，存储全部实际长度为长度为32*（n+1）内存对象
3、每个koblst_t，都包括一个kmsob_t链表...
展开

作者回复: 大佬 66666

最后，想问一下，koblst_t在什么情况下，会挂载多个kmsob_t呢？感觉内存对象不够用，就去补充kmbext_t了吧？

目前不会，因为还没有 区分 kmsob_t的类型

4

pedro
2021-06-21

评论越来越少，证明能跟上的人也越来越少。
作为一个每节都评论，都思考的人，我也能感受到内容难度的加大，而且缺乏阅读方向，代码量越来越大，但是文章要解决什么问题，思路是什么，为什么要这么解决，为什么不

用流程图画一下等等，都能改善专栏阅读的困难。

…

展开

作者回复: “文章要解决什么问题，思路是什么，为什么要这么解决” 文章中都有啊 ，页面分配都懂了， 这个应该很简单才对，有什么地方觉得很难的

3 4

朱熙

2021-06-21

感觉确实如pedro所说，能跟上的人越来越少，可能也和内核越来越深入有关系。
个人想法是不用纠结代码的每一行，更多的跟着注释体会代码整体流程，知道每块代码在做什么，如果有必要，再去看每行代码。
也希望将来能够每个模块与linux内核做一个比较，讲解下优缺点或者差异，让读者能够简单了解linux为了某些目的，增加了那些逻辑。

展开

作者回复: 对的

3

青玉白露

2021-07-13

之所以分配内存大小要采用cache的倍数，主要是为了合理的使用CPU的缓存。
顺带一提，C语言的代码命名确实一个很大的问题，词不达意，很难理解一个函数到底是来干什么的，相比之下，这段时间学习的Java命名格式就显得好的多了，以后我自己写c语言代码的时候就需要借鉴java的命名方法。

展开

作者回复: 因为我搞了很多新的概念 ，那些新的概念你不懂 。也不需要懂，只要明白有什么作用就好了

1

blentle

2021-06-21

提个建议，能否用有意义的名称来命名函数或变量，要不然函数调用关系看的云里雾里.

其实思考题在最开始的几篇提cpu三级缓存的时候已经提了，为了能有效的让缓存生效，提高缓存命中率. 不够的都用padding还是去拼接成缓存行的倍数

作者回复: 你好 是的

6 1

xhy
2021-07-10

希望老师把这些数据结构简写的全英文名称写出来，否则看着这样的命名太难受了

编辑回复: 感谢你的反馈，小编来回应下。因为有些概念是老师自己设计的，比如 mf_olkty 就是 msadsc flags order link type。但是我们又不知道order link type是啥，这个是老师特别设计的属性，方便未来扩展用。所以，咱们学习重点就是知道各种设计的功能，而不是纠结名字哦。

3

Fan
2021-06-29

打个卡，不容易呀。来到这里了。

展开

作者回复: 哈哈 很困难吗